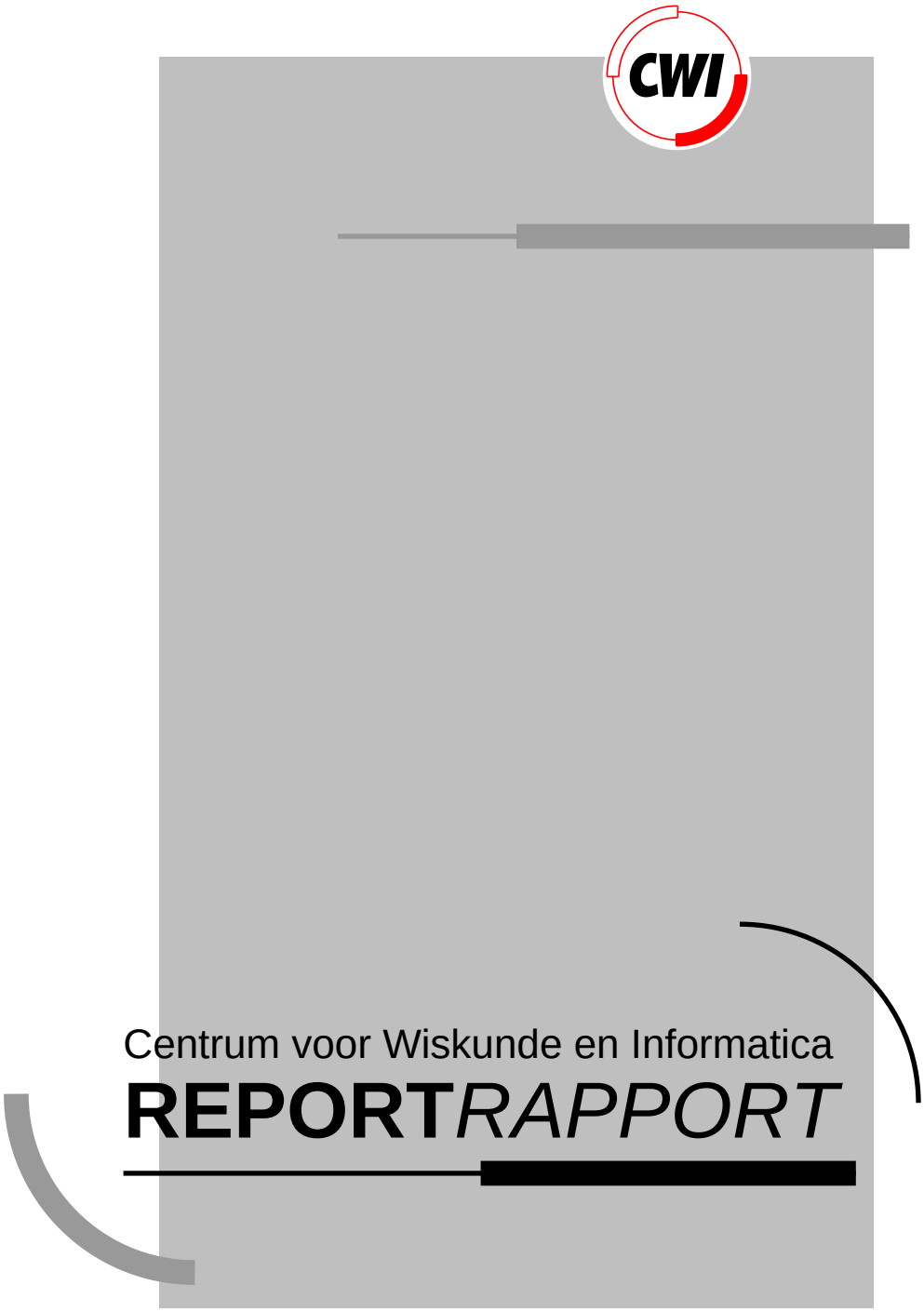

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Fast Parallel Permutation Algorithms

J. Keller

Computer Science/Department of Algorithmics and Architecture

CS-R9303 1993

Fast Parallel Permutation Algorithms

Jörg Keller
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

We investigate the problem of permuting n data items, each covering D global memory cells, on an EREW PRAM with $n/\log n$ processors and less than Dn additional storage. We present a family of algorithms $(\mathcal{A}_k)_k$, where $k = 1, 2, \ldots, \log^* n$, such that $\mathcal{A}_k$ needs time $D \log n \log^{(k)} n$. Here $\log^{(k)} n$ denotes k times application of $\log$ to n, and $\log^* n = \min\{k | \log^{(k)} n \leq 1\}$. All algorithms need $\Theta(n)$ operations which is optimal. The storage requirements for $\mathcal{A}_k$ are n global bits, $n/\log^{(k-1)} n$ global memory cells if $k \geq 2$, and $D \log n / \log^{(k-1)} n$ (resp. D) local memory cells per processor if $k \geq 2$ (resp. $k = 1$). Hence, the family $(\mathcal{A}_k)_k$ reveals a time-space tradeoff. The result can be generalized to machines with $p < n/\log n$ processors such that the run time is $(Dn/p) \log^{(k)} n$, $(Dn/p)/\log^{(k-1)} n$ local cells per processor are needed, and the global storage requirements remain as given above.

1991 Mathematics Subject Classification: 68P05, 68Q10, 68Q22, 68Q25
CR Categories: E.2, F.2.2, G.2.1
Keywords and Phrases: Permutations, Parallel Algorithms, Time-Space Tradeoff
Note: This research is partially supported by the Dutch Science Foundation (NWO) through NFI Project ALADDIN under Contract number NF 62–376.

1 Introduction

Consider the task of permuting n data items, each covering D cells of global memory, on an EREW PRAM with p processors. The task is trivial if there are either Dn additional cells in global memory or Dn/p local memory cells per processor available. Each processor moves n/p items to the additional storage, then it writes these items back in permuted order.

However, if the additional storage is less than Dn, the task gets more difficult. Because not all items can be backed up, at least some of them have to be permuted without additional storage. This can be done by starting with an item x and following the cycle $x, \pi(x), \pi(\pi(x)), \ldots, x$. At most two items have to be backed up, no matter how long the cycle is.

If the permutation is fixed as in [1] or the class of permutations is restricted as in [3], then the structure of the permutation can be used to schedule the work between the processors. Otherwise, we have to keep track about which data items already have been visited, in order to prevent two processors working on the same cycle and permuting some items twice. This simple idea leads to a simple basic algorithm $\mathcal{A}_1$: Each processor P_i takes care for a block of n/p data items. P_i starts with

an unvisited item x of its block and follows the cycle that starts in x until it meets an item x' that is already visited. This might happen because P_i has either reached the start of the cycle again ($x = x'$), or because another processor started at x' to work on the same cycle. P_i now searches for another unvisited item in its block and continues. P_i terminates if all items in its block are visited.

However, this simple algorithm might lead to an unbalanced computation if many processors terminate very early. By interrupting the basic algorithm after a certain number of steps and reallocating processors to work, we can enhance the performance of the basic algorithm. But to do that, we need more additional storage. The time-space tradeoff is exactly described in the main Theorem 1. In the sequel, $\log^{(k)} n$ shall denote k times application of log to n, more exactly $\log^{(1)} n = \log n$, $\log^{(k+1)} n = \log^{(k)}(\log n)$ for $k \geq 1$. The term $\log^* n$ is defined as $\log^* n = \min\{k | \log^{(k)} n \leq 1\}$.

THEOREM 1 *The basic algorithm $\mathcal{A}_1$ takes time $O(D(\log n)^2)$ on an EREW PRAM with $n/\log n$ processors. It needs n additional bits in global memory and $O(1+D)$ local memory cells per processor. It can be enhanced, forming a family of algorithms $(\mathcal{A}_k)_k$ such that $\mathcal{A}_k$ takes time $O(D \log n \log^{(k)} n)$. The new algorithms additionaly need $n/\log^{(k-1)} n$ cells in global memory and $O(D \log n/\log^{(k-1)} n)$ local cells per processor.*

We will describe the basic algorithm in detail in section 2 and prove that it works with $\Theta(n)$ operations on an EREW PRAM with p processors. In section 3 we will show the run time and storage requirements as given in Theorem 1. We will present refinements to the algorithm that result in the whole family of algorithms in section 4, and we will analyze their run time and storage requirements.

2 Basic Algorithm

We want to permute n data items on a p processor PRAM. Let $B = n/p$ (w.l.o.g. we assume n to be a multiple of p). We partition the n items in p blocks of size B such that block i contains items iB to $(i+1)B - 1$. All items are assigned an additional bit that indicates whether they are already visited or not. The bits are initially all set to zero.

A processor can be in one of three states: either it is searching for an unvisited item in its block, or it is working on a cycle, or it is terminated.

If a processor is "searching", it examines the items in its block to test whether they are visited. It continues until it finds an unvisited item or until it reaches the end of the block. In the first case, it marks the item as visited and changes its state to "on cycle". In the latter case, it changes its state to "terminated". A processor never forgets how far it has searched its block yet. If it will change its state from "on cycle" to "searching" again, it does not have to start from the beginning of the block.

If a processor is "on cycle" and has reached item x, then its action depends on the state of x. If the item is not yet visited, then the processor will pick up the item, mark it as visited, store in x the item it picked up in the previous step, and move to $\pi(x)$. If the item is already visited, then the processor will store the previous item in x and change its state to "searching". If a processor just has changed its state to "on cycle", then it may not mark the item as visited, because it already marked it, and it has no previous item to store in x.

A processor may meet a visited item either because it reaches the end of the cycle (the item where it started in its own block) or because another processor started to work on the same cycle in this item.

An item x in block i can now be marked visited by two events: either processor i searching block i finds x to be unvisited, or any processor following the cycle on which x lies reaches x. In order to

keep the algorithm correct, we have to avoid a conflict between these two cases. Therefore, we split each step of the algorithm in two parts such that "searching" processors and processors "on cycle" make steps alternately.

The program for the basic algorithm is shown in figure 1. There, T denotes the maximum number of steps to take. We will compute an upper bound for T in section 3. Each processor has local variables `state`, `index`, `item`, `buffer` and `buffer2`. Variable `state` defines the current state of the processor, `index` counts how far a processor has searched its block, `item` points to the currently visited item, and `buffer` and `buffer2` are used to store data items temporarily. Global arrays are `visited` and `item`. Array `visited` contains the flags of all items, `item` contains the items themselves.

Note that in order to improve the constant factor in run time, also the processors "on cycle" search in the first part of a step.

Each item can be touched at most twice, once by the processor that owns the block to which the items belongs, and once by any processor following the cycle on which the item is. Hence the total number of operations is $\Theta(n)$.

Concurrent access to an item could only happen if its owner marks it and in the same step, another processor following a cycle meets it, too. By spliting each step into two parts, this is avoided and therefore an EREW PRAM is sufficient.

3 Analysis

In order to simplify the analysis we will assume that each processor can find the next unvisited item in its block in unit time. The time actually spent on searching the block is at most B and does not affect the worst case behaviour sketched below. Therefore we will neglect it.

Consider the algorithm permuting n items with p processors such that $p > B$. We assume $D = 1$ and $B \geq 2$ for simplicity. The run time is largest if each processor terminates as early as possible, because then the work done per step is minimal, and with a fixed work to do, the run time is largest. This worst case can be achieved by a permutation, where all processors work in as few blocks as possible. Then in one step p/B blocks can be visited completely and the respective processors terminate. If p_i processors are still active after step i, it follows that $p_{i+1} \geq p_i - p_i/B = p_i(1 - 1/B)$.

If we define $p_0 = p$ then $p_i \geq p(1 - 1/B)^i$. This continues as long as $p_i \geq B$. Let k be the maximal index such that $p_k \geq B$. Then, after $k + 1$ steps, the remaining $p_{k+1} < B$ processors need B/p_{k+1} steps to visit one block completely. The processor that took care of that block terminates and $p_{k+1} - 1$ processors remain. They take $B/(p_{k+1} - 1)$ steps to visit the next block completely. This continues until all the remaining p_{k+1} blocks are visited.

The total run time is $T = k + \sum_{i=1}^{p_{k+1}} B/i \leq k + B \ln p_{k+1} \leq k + B \ln B$.

In order to compute k, we assume the worst case $p_k = p(1 - 1/B)^k$ and find an upper bound on k such that $p_k \geq B$. The inequality $p(1 - 1/B)^k \geq B$ can be solved to $k \leq \log(B/p)/\log(1 - 1/B) = \log(p/B)/\log(B/(B - 1))$.

With $\log(z + 1) - \log(z) \geq 1/z$ we obtain $k \leq \log(p/B) \cdot (B - 1)$ and hence

$$T \leq B(\log(p/B) + \ln B) = O((n/p) \log n)$$

since $B = n/p$. The total run time for the case $p = n/\log n$ is at most $T = O((\log n)^2)$.

```
(* Initialization *)
for i := 0 to p − 1 pardo
  P_i.state := SEARCHING ;
  P_i.index := 0 ;
  for j := 0 to B − 1 do
    visited[iB + j] := 0
  od
od ;

for t := 1 to T do
  for i := 0 to p − 1 pardo
    (* first part of step *)
    if P_i.state ≠ TERMINATED then
      (* first part of step *)
      if P_i.state = SEARCHING then
        P_i.item := iB + P_i.index
        if visited[P_i.item] = 0 then
          visited[P_i.item] := 1 ;
          P_i.state := NEW CYCLE ;
          P_i.buffer := item[P_i.item]
        else
          P_i.index := P_i.index + 1 ;
          if P_i.index = B then
            P_i.state := TERMINATED
          fi
        fi
      else
        if visited[Bi + P_i.index] = 1 then
          P_i.index := P_i.index + 1 ;
          ifP_i.index = B then
            P_i.state := TERMINATED
          fi
        fi
      fi ;
      (* second part of step *)
      if P_i.state = ON CYCLE then
        if visited[P_i.item] = 0 then
          visited[P_i.item] = 1 ;
          P_i.buffer2 := item[P_i.item] ;
          item[P_i.item] := P_i.buffer ;
          P_i.buffer := P_i.buffer2 ;
          P_i.item := π(P_i.item)
        else
          item[P_i.item] := P_i.buffer ;
          P_i.state := SEARCHING ;
          P_i.index := P_i.index + 1
        fi
      else
        if P_i.state = NEW CYCLE then
          P_i.item := π(P_i.item) ;
          P_i.index := ON CYCLE
        fi
      fi
    fi
  od
od ;
```

FIGURE 1. Basic Algorithm

The fact $\log(z+1) - \log(z) \geq 1/z$ can be obtained by observing that $\log(x)$ is continuous and differentiable and that its first derivative $1/(x \ln 2)$ in the interval $[z, z+1]$ is always larger than $1/z$.

If $D > 1$ it is straightforward that $T \leq O((Dn/p) \log n)$.

From the description of the algorithm it is clear that it needs n global bits and that $O(D)$ local memory cells per processor are sufficient.

4 Refinements

The major drawback of algorithm $\mathcal{A}_1$ is that many processors could terminate very early. If these idle processors could be used again to do useful work, the run time could be decreased. To do that, we have to interrupt the basic algorithm before too many processors have terminated, and to redistribute work to processors.

Assume we interrupt the basic algorithm after t steps. For simplicity, we again consider $D = 1$. In step i at least p_i processors are active that visit data items. Hence, after t steps, at least $S_t = \sum_{i \leq t} p_i$ data items are visited. With $p_i \geq p(1 - 1/B)^i$ we obtain

$$S_t \geq p \sum_{i \leq t} (1 - 1/B)^i = p \frac{1 - (1 - 1/B)^{t+1}}{1 - (1 - 1/B)} = pB(1 - (1 - 1/B)^{t+1})$$

With $n = pB$, the number of unvisited items after t steps accounts to $R_t = n - S_t \leq n(1 - 1/B)^{t+1}$. Observing that $(1 - 1/x)^x \leq 1/e$ leads to $R_t \leq n(1/e)^{(t+1)/B}$. Assume that $t = B \ln Z - 1$ for some Z. Then $R_t \leq n/Z$.

If $Z = B$, then at most $n/B = p$ items remain unvisited. If each processor can be assigned to one of those items in time t, then the processors could pick up the remaining items and store them in permuted order in time $O(D)$. This leads to algorithm $\mathcal{A}_2$, which runs in time $O(tD)$. For $p = n/\log n$, this yields a run time of $O(tD) = O(D \log n \log\log n)$. The storage requirements are n global bits, $O(D)$ local memory cells per processor and the storage needed for re-assigning the processors.

The algorithms $\mathcal{A}_k$, $k \geq 3$, are obtained by choosing $Z = \log^{(k-2)} B$. In case $p = n/\log n$, $n/\log^{(k-1)} n$ items remain unvisited. Each processor picks up $\log n/\log^{(k-1)} n$ of them and stores them in permuted order. This leads to run times $D \log n \log^{(k)} n$ and increases the storage requirements to $DR_t/p = D \log n/\log^{(k-1)} n$ local memory cells per processor.

k can be at most $\log^{(*)} n$, where an optimal run time of $\log n$ is reached but where storage requirements reach $D \log n$ local memory cells per processor. This is no better than the trivial algorithm mentioned in the introduction.

The number of operations for all algorithms still is $\Theta(n)$.

All algorithms can be slowed down using $p < n/\log n$ processors. This can be done by simulating $V = n/(p \log n)$ "virtual" processors of the $n/\log n$-processor algorithm on the p-processor machine. The run time increases by a factor of V, resulting in $T = O((Dn/p) \log^{(k)} n)$. The global storage requirements remain as they are. The local storage requirements are also increased by a factor V, leading to $(Dn/p) \log^{(k-1)} n$ local memory cells per processor.

We finish the section by presenting how processors are re-assigned to unvisited items. First, the

processors produce a list of indeces of unvisited items. The list has length $l = n/\log^{(k-1)} n$. Processor i will take care for items il/p to $(i+1)l/p - 1$ in that list.

The list is generated with the help of parallel prefix. Each processor i counts the number u_i of unvisited items in its block in time $O(B)$. The values u_i serve as the operands of the parallel prefix operation. Each processor i gets back $U_i = \sum_{j<i} u_j$. The parallel prefix operation takes time $O(\log p) = O(\log n)$ on an EREW PRAM with p processors and requires $O(p) = O(n/\log n)$ cells in global memory [2]. U_i gives processor i an index in the list where it can store the indices of the unvisited items in its block. This again takes time $O(B)$. Hence, the re-assignment takes time $O(B + \log n)$ on an EREW PRAM and requires $n/\log^{(k-1)} n$ global memory cells.

5 Concluding Remarks

In our analysis, we have distinguished between bits, numbers, and items. Bits are considered different, because implementing them often will not increase the storage at all. In the items' representations, there will be often an unused bit that can be used to encode the "visited" bits. Also, many memory subsystems today provide additional bits per cell or record that are used for parity etc. One of them could probably be used for implementing the bits. We distinguish between numbers and items because in many cases, D could be very large, for example if each item is a memory page. As the extra global storage needed covers a fraction of $1/(D \log n)$ of the items, this will be very small for large D.

If we can implement the bits in the items' structure, then the most simple algorithm $\mathcal{A}_1$ has the advantage that it needs no other global storage. This could make it helpful if the complete global memory has to be permuted as is the case when rehashing the complete address space of a PRAM emulation.

$\mathcal{A}_1$'s worst case behaviour depends on the permutation. For many permutations the behaviour should be much better than $D(\log n)^2$. We support this assumption by simulation results. For $n = 2^i$, $2 \le i \le 17$, we simulated the algorithm on 100 randomly chosen permutations. The average run times and the variations can be seen in figure 2. All variances are very small, the run times are all smaller than $3 \log n$. This hints at $\mathcal{A}_1$ having a much better average case behaviour than worst case behaviour. However, the average case run time of $\mathcal{A}_1$ still has to be analyzed.

Acknowledgements

The author wants to thank Dany Breslauer and John Tromp for their patience to listen and their helpful suggestions.

References

[1] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. On communication latency in PRAM computations. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21. ACM, 1989.

[2] Richard M. Karp and Viaya L. Ramachandran. A survey of parallel algorithms for shared–memory machines. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A*, pages 869–941. Elsevier, 1990.

[3] Jörg Keller. Hashing and rehashing in emulated shared memory. Report CS–R9240, Centrum voor Wiskunde en Informatica, 1098 SJ Amsterdam, The Netherlands, 1992.

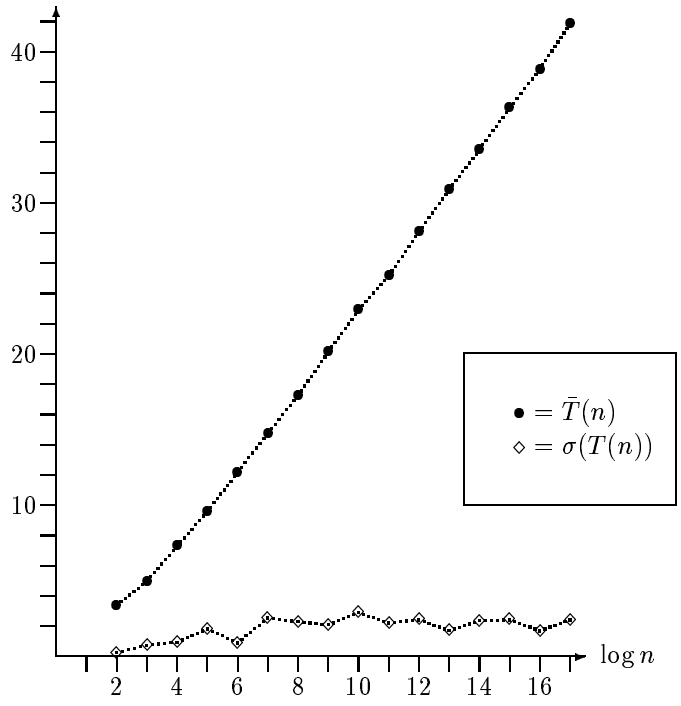

FIGURE 2. Average run time of $\mathcal{A}_1$ and variation